CompSci 330 Assignment 2

Implement a simple language to generate logic circuits

Obtain the ASSIGN2STUD directory from the assignment web pages, as the basis for your assignment. Modify the code for the interpreter.

Overview

In the computer industry, it is quite common to have text files that provide information for a program. These text files can be lexically analysed and parsed, and used to create a data structure that is then processed in some way, to perform some task.

For example, we might have a configuration file to specify how devices are to be set up, and mounted when a computer system is booted. We might have a text file that describes the layout of dialog windows for a GUI interface. We might obtain a text file as the result of running an SQL query, and want to convert the information into HTML for display via a browser. We might want to take a formula in a spreadsheet document, lexically analyse it and parse it to build a tree, then perform a treewalk of the tree to evaluate the formula. There are many applications that perform similar tasks to compilers that are not normally thought of implementing a computer language.

The CPU of a computer is a “logic circuit” composed of “components” (such as “and”, “or”, “xor”, and “not” gates, clocks, etc), and connecting “paths” (wires). A simple path can be thought of as a Boolean variable (in other words, a “bit”, perhaps represented by a high or low voltage). A component has input paths, output paths and perhaps some internal state. A simple component can be thought of as a thread that loops. Each time through the loop, it waits for an input to change, then alters its outputs appropriately. Paths may be grouped together to form path arrays. Components and paths may be grouped together to form compound components.

It is possible to create a language for describing logic circuits. We could run a program written in this language to create masks for use in the fabrication of computer chips. Alternatively, we could create threads to represent the elementary components, and objects to represent the connecting paths, then start the threads executing to “simulate” the logic circuit. That is what we are going to do. It is not hard, but you need to know a little bit about logic circuits. Many years ago, Bob Doran wrote a wonderful little book, “Introduction to logic circuits using the logic simulator”. It is available in the university library with classification number 511.3 D69. There are other more recent books that provide similar information, for example, the Physics 243 text, “Digital Fundamentals” by T.L. Floyd. However, these books might not have the logic circuits explicitly written in pseudocode.

So, what is this language like? It is built on top of a traditional computing language, with integer, Boolean and string variables and expressions, control statements, and function declarations and invocations. It could do with more traditional features, especially more ability to structure data and code, but these are sufficient for our purpose.

The execution of a program written in this computer language “creates” the components and connecting paths that represents a logic circuit (i.e., creates suitable Java objects to simulate components and connecting paths). It does not itself simulate the execution of the logic circuit. A “for” statement can be used to create multiple copies of a logic circuit (essentially an array of components). An “if” statement can be used to create one of two alternative logic circuits. A function declaration defines a compound component in terms of other components and paths. A function invocation creates an instance of a component, and connects it to its parameter input and output paths.

As well as simple variables, it is possible to declare simple path variables, and arrays of path variables. It is also possible to group and ungroup the elements of path arrays. However, path variables cannot be used in expressions. They only have values when the logic circuit is simulated, something that occurs after the program is executed.

A function represents a component. Functions take input path arrays, values, and output path arrays as parameters. The value parameters are used to specify such things as the size of a path array or memory, etc. Functions may have local integer, Boolean and string variables and path variables. Built-in functions create elementary components. Elementary components and paths created within a function maintain their existence, even after the function returns. Unfortunately, functions do not return a value (a nice extension for the future).

A typical function declaration has the form

```
component { in PathDefnList } IDENT ( ValueParamDefnList ) { out PathDefnList  }
begin
    LocalDeclStmtList
end
```

However some portions may be omitted. The "IDENT" represents the name of the function. "{ in PathDefnList }" represents the input path array parameters, "(ValueParamDefnList)" the integer, Boolean and string parameters, and "{ out PathDefnList }" the output path array parameters.

A simple function invocation has the form

```
 { in PathArrayList } IDENT ( ExprList ) { out PathArrayList };
```

Again some portions may be omitted. Also function invocations may be collapsed together if the output paths from an invocation are the same as the input paths for the next invocation.

Built-in (Library) Functions (Components)

From now on we will call functions "components".

There are a number of built-in (library) component declarations, declared in a file "LIBRARY/globalLibrary.in", that is implicitly included in every program. No body is specified for a built-in component (because it is "built-in").

The library components are:

- Components to represent "and", "or", "xor" (exclusive or), "not" and "join" gates.

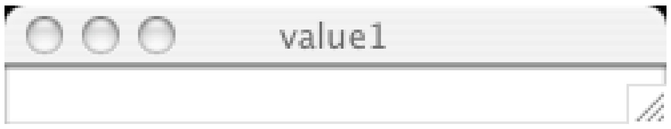
  ```
  component { in opd[ n ] } or( n ) { out result };
  component { in opd[ n ] } and( n ) { out result };
  component { in opd[ n ] } xor( n ) { out result };

  component { in opd[ n ] } not( n ) { out result[ n ] };
  component { in opd[ n ] } join( n ) { out result[ n ] };
  ```

  These represent the building blocks from which "combinational circuits" can be constructed. The integer parameter "n" represents number of bits in the path array "opd", and perhaps "result". The "or" component sets the value of "result" to the "or" of the bits that make up "opd". The "and" and "xor" components are similar. The "not" component sets the value of "result" to the one's complement of "opd". The "join" component just sets the value of "result" to the value of "opd". Combinational circuits are used to create components that perform arithmetic, etc.

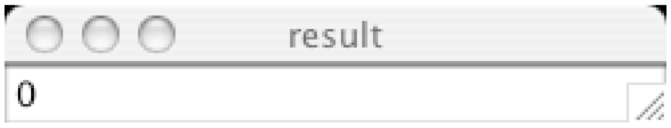
- A component to represent a literal constant.

  ```
  component literal( n, litValue ) { out result[ n ] };
  ```

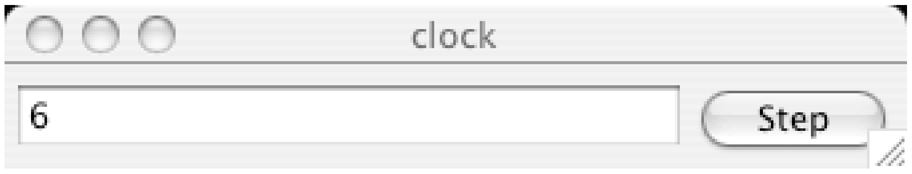
  The binary representation of the literal value is used to specify the values of the "n" bits that make up the "result".

- A GUI component with an editable text field used to provide input to a logic circuit, in the form of an integer value.
```
component input( name, x, y, base, n ) { out result[ n ] };
```


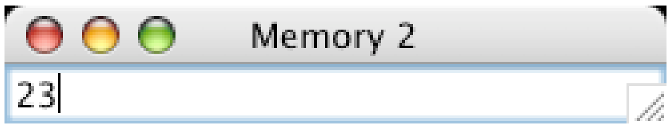
The GUI component is displayed with title "name", at coordinates "(x, y)". Note that you must type return after editing the text field to activate the component. The text in the text field is parsed as an integer value in the specified base. The binary representation of the integer value is used to specify the values of the "n" bits that make up the "result".

- A GUI component with a text field to display the output of a logic circuit as an integer value.
```
component { in opd[ n ] } output( name, x, y, base, n );
```


The GUI component is displayed with title "name", at coordinates "(x, y)". The value of "opd" interpreted as the binary representation of the integer value is displayed in the specified base.

- A GUI component with an editable text field and button used to provide a value that can be incremented by pressing the button. This component can be used to represent a "clock".
```
component counter( name, x, y, base, n ) { out result[ n ] };
```


The binary representation of the integer value typed into the text field is used to specify the values for "result". The value is incremented modulo 2^n each time the button is pressed. This component could do with additional buttons to allow it to step automatically.

- A component used to represent an "n" bit memory, that has a GUI component with an editable text field to display the value.
```
component
     { in read, write, init, visible, opd[ n ] }
          memory( name, x, y, base, n, initValue )
          { out result[ n ] };
```


The initial value of the memory is "initValue". If "visible" is set, the GUI component is displayed with title "name", at coordinates "(x, y)". The internal state may be altered by editing the text field. If "read" is true, "result" is set to the value of the memory. If "write" is true, the value of the memory is set to "opd". If "init" is true, the value of the memory is set to "initValue".

- A component that takes an integer value represented by "opd", and sets the corresponding bit in "result" to true, and all other bits in "result" to false.
```
component { in opd[ m ] } decode( m ) { out result[ 1 << m ] };
```
The decode component provides the logic circuit equivalent of indexing.

- A component that, based on an m-bit integer value represented by “index”, selects “n” bits from “alternative”, starting at position “n * index”, and sets “result” to this value.

  ```
  component
       { in index[ m ], alternative[ ( 1 << m ) * n ] }
            select( m, n )
            { out result[ n ] };
  ```

  The select component provides the logic circuit equivalent of an “if” or “switch” statement.

- Two paths, “set” and “clear” are also built in, and have the values true and false.

  ```
  path clear, set;
  literal( 1, 0 ) { out clear };
  literal( 1, 1 ) { out set };
  ```

In fact, only the gate components and GUI components are really needed. The decode and select components could be built from gates.

Some Examples

Example 1 Addition

We can use these built-in logic components to declare a compound component that performs addition.

```
/****************************
add.lib
****************************/
```

A “half adder” is a logic circuit that takes two binary digits, and computes their sum (the “exclusive or” of the bits), and the carry (the “and” of the bits). For example, 1 + 0 = 1, with carry 0, and 1 + 1 = 0, with carry 1.

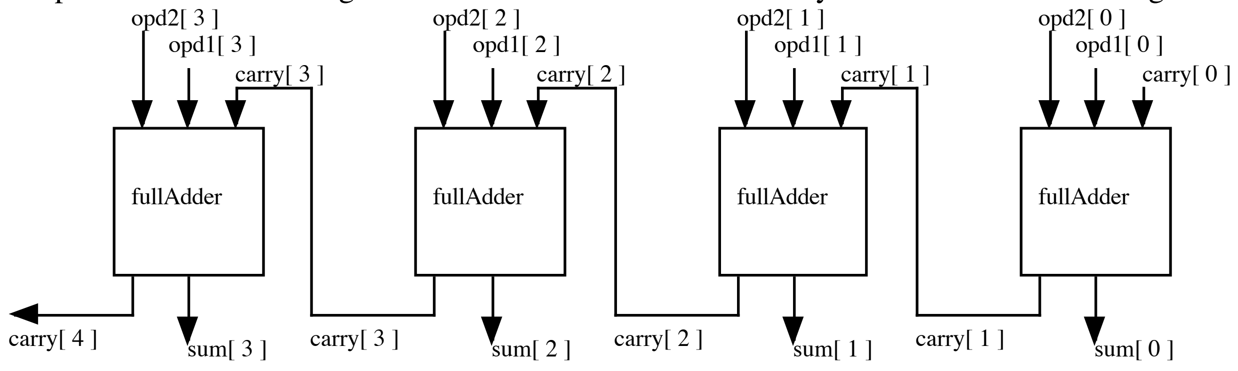
```
component { in opd1, opd2 } halfAdder { out sum, carry }
     begin
          { in opd1 opd2 } xor( 2 ) { out sum };
          { in opd1 opd2 } and( 2 ) { out carry };
     end
```

Two half adders can be combined to produce a “full adder”, that takes two binary digits, together with a carry in, and generates the sum and carry out. For example, if we have a carry in of 1, and add 1 + 1, we get 1, with a carry out of 1.

```
component { in opd1, opd2, carryIn } fullAdder { out sum, carryOut }
     begin
          path sum1, carry1, carry2;
          { in opd1, opd2 } halfAdder { out sum1, carry1 };
          { in sum1, carryIn } halfAdder { out sum, carry2 };
          { in carry1 carry2 } or( 2 ) { out carryOut };
     end
```

By combining an array of “n” full adders, we can add two “n” bit numbers. The carry out from adding the “i”th bits becomes the carry in when adding the “i+1”th bits, so the carry ripples through the circuit, and the component is called a “ripple adder”. The algorithm executes in O(n) time.

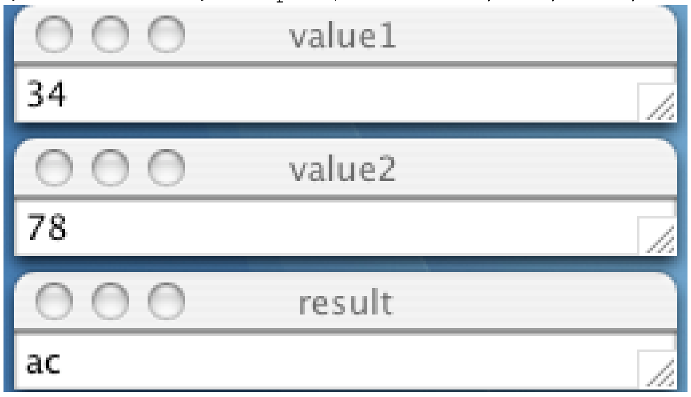
```
component { in opd1[ n ], opd2[ n ], carryIn } add( n )
     { out sum[ n ], carryOut }
     begin
          path carry[ n + 1 ];
          { in carryIn } join( 1 ) { out carry[ 0 ] };
          for i from 0 upto n do
               { in opd1[ i ], opd2[ i ], carry[ i ] } fullAdder
                    { out sum[ i ], carry[ i + 1 ] };
          end
          { in carry[ n ] } join( 1 ) { out carryOut };
     end
```



We can use the built-in GUI components to test this code.

We create two input GUI components to provide the operands, and an output GUI component to display the result. We connect them together via an "add" component that does the addition. We also make the initial carry in 0.

```
/*****************************
add
*****************************/

include "LIBRARY/add.lib";

define base = 16, n = 8;

path value1[ n ], value2[ n ], carryIn, result[ n ], carryOut;

input( "value1", 10, 50, base, n ) { out value1 };
input( "value2", 10, 100, base, n ) { out value2 };

literal( 1, 0 ) { out carryIn };

{ in value1, value2, carryIn } add( n ) { out result, carryOut };
{ in result } output( "result", 10, 150, base, n );
```



34

78

ac

Example 2 Comparison

We can compare two operands, to determine whether one is <, ==, or > than the other.

```
/*****************************
compare.lib
*****************************/
```

First we define a simple component to compare bits.

The notation "xor(2).not(1)" means invoke "xor(2)" and pass its output to "not(1)". In other words, "equal = ! (opd1 ^ opd2)". Paths may be grouped together to form a path array by juxtaposing their names, as in "opd1 opd2".

```
component { in opd1, opd2 } compareBit { out less, equal, greater }
     begin
          path notOpd1, notOpd2;
          { in opd1 } not( 1 ) { out notOpd1 };
          { in opd2 } not( 1 ) { out notOpd2 };
          { in notOpd1 opd2 } and( 2 ) { out less };
          { in opd1 opd2 } xor( 2 ).not( 1 ) { out equal };
          { in notOpd2 opd1 } and( 2 ) { out greater };
     end
```

Then we define a component to compare "n" bits, that recursively splits the task up into comparing the two halves. This is better than iterating through the bits, because the two halves really do execute in parallel, and hence the algorithm executes in O(log n) time.

The notation "opd[size @ index]" means the subarray of "size" bits, of the path array "opd", starting at index "index".

```
component { in opd1[ n ], opd2[ n ] } compare( n ) { out less, equal, greater }
     begin
          if n == 1
          then
               { in opd1[ 0 ], opd2[ 0 ] } compareBit
                    { out less, equal, greater };
          else
               path lessLow, equalLow, greaterLow;
               path lessHigh, equalHigh, greaterHigh;
               path less1, greater1;
               { in opd1[ n / 2 @ 0 ], opd2[ n / 2 @ 0 ] }
                    compare( n / 2 )
                    { out lessLow, equalLow, greaterLow };
               { in opd1[ n - n / 2 @ n / 2 ], opd2[ n - n / 2 @ n / 2 ] }
                    compare( n - n / 2 )
                    { out lessHigh, equalHigh, greaterHigh };
               { in equalHigh lessLow } and( 2 ) { out less1 };
               { in equalHigh greaterLow } and( 2 ) { out greater1 };
               { in lessHigh less1 } or( 2 ) { out less };
               { in equalHigh equalLow } and( 2 ) { out equal };
               { in greaterHigh greater1 } or( 2 ) { out greater };
          end
     end
```

We can use the built-in GUI components to test this code.

```
/*****************************
compare
*****************************/

include "LIBRARY/compare.lib";

define base = 16, n = 8;

path opd1[ n ], opd2[ n ], less, equal, greater;

input( "opd1", 10, 100, base, n ) { out opd1 };
input( "opd2", 10, 150, base, n ) { out opd2 };

{ in opd1, opd2 } compare( n ) { out less, equal, greater };
{ in less } output( "less", 10, 200, base, 1 );
{ in equal } output( "equal", 10, 250, base, 1 );
{ in greater } output( "greater", 10, 300, base, 1 );
```

Example 3 Left Shift

We can also create a compound component to perform a left shift of one operand by an amount specified by another operand.

```
/****************************
shift.lib
****************************/
```

The “shiftIf” component takes an “n” bit value “opd”, and either performs no shift, if “cond” is false, or shifts the value by “p” bits.

```
component { in cond, opd[ n ] } shiftIf( n, p ) { out result[ n ] }
     begin
          path zero[ p ];
          literal( p, 0 ) { out zero };
          { in cond, opd[ n - p @ 0 ] zero opd } select( 1, n ) { out result };
     end
```

The “shift” component takes an “n” bit value “opd”, and shifts it by the number of bits represented by the value of “shiftBy”. It does this by an algorithm, that looks at successive bits in “shiftBy”. If the “i”th bit is 1, it performs a shift by “2^i” bits.

This algorithm executes in O(m) time.

```
component { in shiftBy[ m ], opd[ n ] } shift( n, m ) { out result[ n ] }
     begin
          if m == 0
          then
               { in opd } join( n ) { out result };
          else
               path result1[ n ];

               { in shiftBy[ m - 1 ], opd }
                    shiftIf( n, 1 << ( m - 1 ) ) { out result1 };
               { in shiftBy[ m - 1 @ 0 ], result1 }
                    shift( n, m - 1 ) { out result };
          end
     end
```

We can use the built-in GUI components to test this code.

```
/****************************
shift
****************************/

include "LIBRARY/shift.lib";

define base = 16, m = 3, n = 8;

path shiftBy[ m ];
path opd[ n ];
path result[ n ];

input( "opd", 10, 100, base, n ) { out opd };
input( "shiftBy", 10, 150, base, m ) { out shiftBy };

{ in shiftBy, opd } shift( n, m ) { out result };
{ in result } output( "result", 10, 200, base, n );
```

Example 4 Flip-Flops and Memory

This example is much more sophisticated, so don't worry if you don't understand it.

We can create what is called a "flip-flop" to store a "bit" (binary digit). This is a logic circuit that has feedback (cycles in the directed graph of components and paths) that provides an internal state.

```
/****************************
flipFlop.lib
****************************/
```

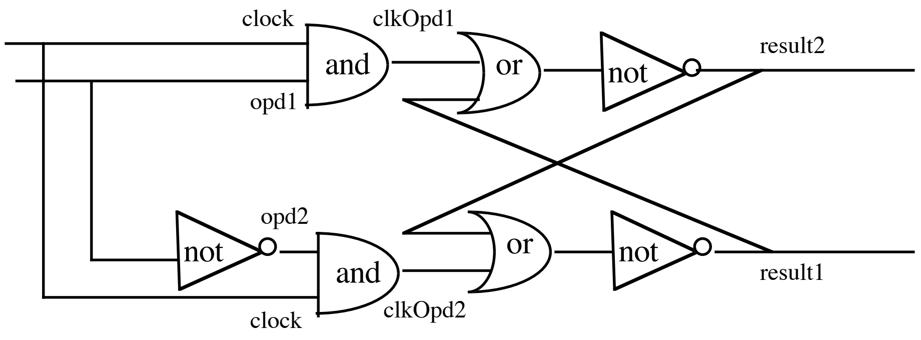
A simple flip-flop takes a clock signal "clock", and a value "opd1" as inputs, and produces a value "result1" as output.

If "clock == true", and "opd1 == true", then "result2 = false", and "result1 = true". If "clock == true", and "opd1 == false", then "result1 = false", and "result2 = true". So if "clock == true", "result1 = opd1", and "result2 = !opd1".

If "clock == false", then "result1" and "result2" can take any value, so long as "result2 == !result1".

So, when the clock is set, a simple flip-flop stores the value of "opd1" in "result1". The value remains there, even after the clock is cleared, and "opd1" changes.

```
component { in clock, opd1 } simpleFlipFlop { out result1 }
     begin
          path opd2, clkOpd1, clkOpd2, result2;
          { in opd1 } not( 1 ) { out opd2 };
          { in clock opd1 } and( 2 ) { out clkOpd1 };
          { in clock opd2 } and( 2 ) { out clkOpd2 };
          { in clkOpd1 result1 } or( 2 ).not( 1 ) { out result2 };
          { in clkOpd2 result2 } or( 2 ).not( 1 ) { out result1 };
     end
```



To allow the input data to stabilise before the change is visible to the output, and to avoid problems when the output feeds back to the input, it is best to pair two simple flip-flops, to form a "master-slave flip-flop". When "clock1" is true, the value of "opd" is transferred to the internal state "value", but does not pass through to the output "result". When "clock1" is false, the internal state "value" is transferred to "result", but changes in the input have no effect. Thus the master-slave flip-flop appears to transfer the data from "opd" to "result" when "clock1" changes from true to false.

```
component { in clock1, opd } masterSlaveFlipFlop { out result }
     begin
          path clock2, value;
          { in clock1 } not( 1 ) { out clock2 };
          { in clock1, opd } simpleFlipFlop { out value };
          { in clock2, value } simpleFlipFlop { out result };
     end
```

We can create an array of flip-flops, representing an "n" bit memory.

```
component { in clock, opd[ n ] } bitMemory( n ) { out result[ n ] }
     begin
          for i from 0 upto n do
               { in clock, opd[ i ]  } masterSlaveFlipFlop { out result[ i ] };
          end
     end
```

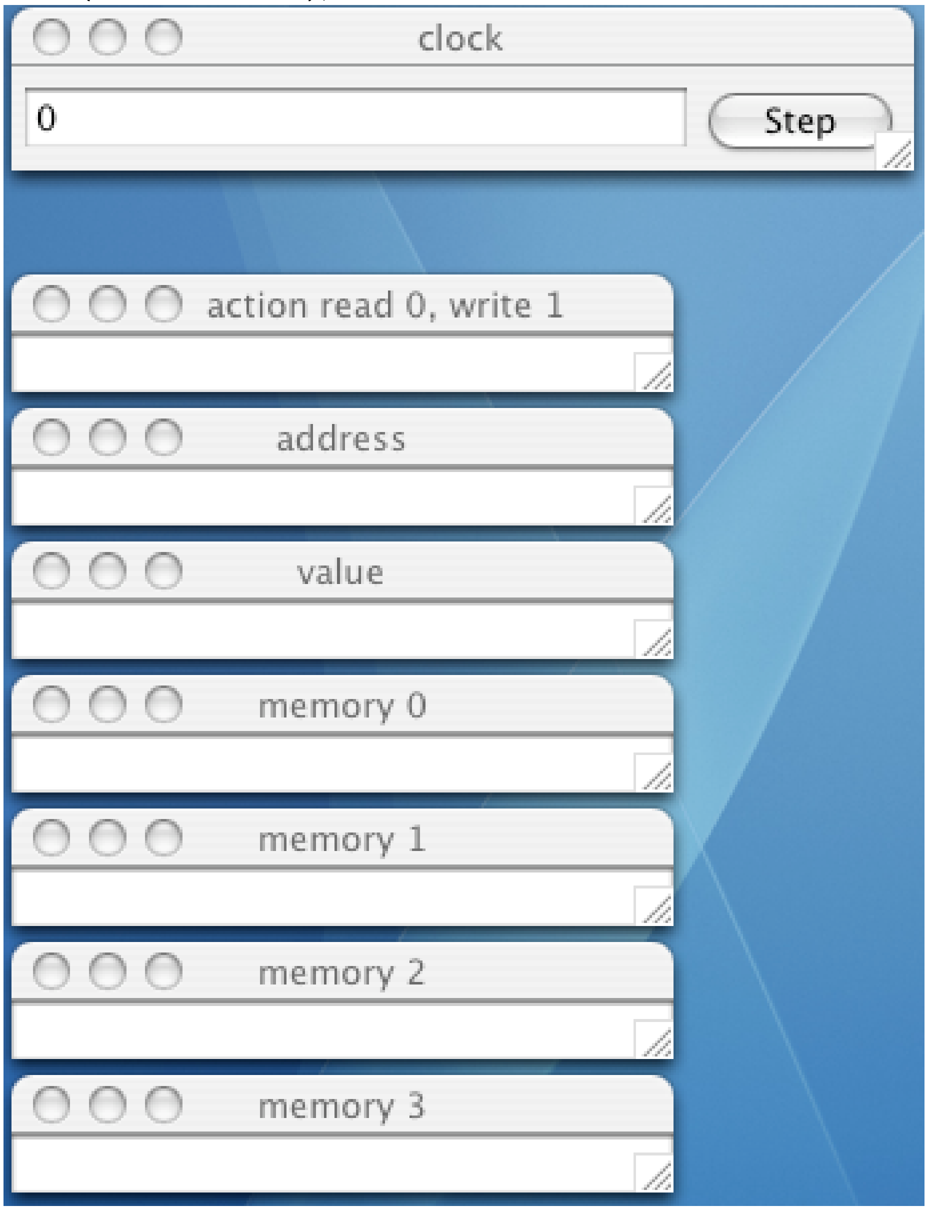
We can use the built-in GUI components to test this code.

```
/****************************
flipFlop
****************************/

include "LIBRARY/flipFlop.lib";

define base = 16, n = 8;

path clock;
path opd1[ n ];
path result[ n ];

input( "opd1", 10, 100, base, n ) { out opd1 };
counter( "clock", 10, 200, base, 1 ) { out clock };
{ in clock, opd1 } bitMemory( n ) { out result };
{ in result } output( "result", 10, 300, base, n );
```

Example 5 Memory Access

It is possible to build higher level logic circuits, with multiple memories and connecting data paths to permit us to load a value into a "register" or store the value of the register into a memory. We can use the decode component to set a flag to specify whether a memory should be written to.

```
/****************************
memory.lib
****************************/

component
     { in read, write, init, address[ m ], opd[ n ] }
          memoryArray( name, x, y, base, m, n )
          { out result[ n ] }
     begin
          path selection[ 1 << m ];
          path readSelect[ 1 << m ];
          path writeSelect[ 1 << m ];
          path resultSelect[ ( 1 << m ) * n ];
          { in address } decode( m ) { out selection };
          for i from 0 upto 1 << m do
               { in read selection[ i ] } and( 2 ) { out readSelect[ i ] };
               { in write selection[ i ] } and( 2 ) { out writeSelect[ i ] };
               { in readSelect[ i ], writeSelect[ i ], init, set, opd  }
                    memory( name & " " & i, x, y + 50 * i, 16, n, 0 )
                    { out result };
          end
     end

/****************************
memory
****************************/

include "LIBRARY/memory.lib";

define base = 16, n = 8, m = 2;

path clock;
path action;
path read, write;
path load, store;
path dataIn[ n ], dataOut[ n ];
path address[ m ];

counter( "clock", 10, 50, base, 1 ) { out clock };

input( "action read 0, write 1", 10, 150, base, 1 ) { out action };
{ in action } decode( 1 ) { out write read };
{ in clock write } and( 2 ) { out store };
{ in clock read } and( 2 ) { out load };

input( "address", 10, 200, base, m ) { out address };

{ in store, load, clear, set, dataOut }
     memory( "value", 10, 250, base, n, 0 )
     { out dataIn };

{ in load, store, clear, address, dataIn }
     memoryArray( "memory", 10, 300, base, m, n )
     { out dataOut };
```



Details of the language

A logic program

A logic program is a list of global declarations and statements.

Declarations and Statements

The different kinds of declarations and statements are:

- Include statements, for example
```
include "LIBRARY/shift.lib";
```
  Include statements cause a sub-file to be processed, as if the include statement was replaced by the content of the file.

  The file “LIBRARY/globalLibrary.in” is implicitly included.

- Value declarations, for example
```
define base = 16, n = 8, m = 2;
```
  Only simple integer, Boolean and string variables can be declared. Moreover the variable has its value defined within the declaration, and cannot be altered (there are no assignment statements in this language).

- Path declarations, for example
```
path opd2, clkOpd1, clkOpd2, result2;
path result[ n ];
```
  Path variables can be declared as simple paths, or path arrays. A simple path is really just a path array of size 1. The size is an (integer) expression.

- If statements, for example
```
if m == 0
then
     { in opd } join( n ) { out result };
else
     path result1[ n ];
     { in shiftBy[ m - 1 ], opd } shiftIf( n, 1 << ( m - 1 ) )
          { out result1 };
     { in shiftBy[ m - 1 @ 0 ], result1 } shift( n, m - 1 )
          { out result };
end
```
  The then and else parts are local blocks of declarations and statements. The else part may be omitted, and it is possible to combine if statements by the use of “elif”, with a single “end” for the complete statement.

- For statements, for example
```
for i from 0 upto n do
     { in clock, opd[ i ]  } masterSlaveFlipFlop { out result[ i ] };
end
```
  The body is a local block of declarations and statements. The loop variable is local to the body, and takes values from that of the initial expression up to but not including the final expression.

- Component declarations, for example
```
component { in opd0[ n ], opd1[ n ] } orBits( n ) { out result[ n ] }
     begin
          for i from 0 upto n do
               { in opd1[ i ] opd0[ i ]  } or( 2 ) { out result[ i ] };
          end
     end
```

- Invocation statements, for example

  ```
  { in opd1 } not( 1 ) { out opd2 };
  { in clock opd1 } and( 2 ) { out clkOpd1 };
  { in clock opd2 } and( 2 ) { out clkOpd2 };
  { in clkOpd1 result1 } or( 2 ).not( 1 ) { out result2 };
  { in clkOpd2 result2 } or( 2 ).not( 1 ) { out result1 };
  ```

  Note the way the last two invocation statements actually combine two invocations. For example

  ```
  { in clkOpd1 result1 } or( 2 ).not( 1 ) { out result2 };
  ```

  is equivalent to

  ```
  path temp;
  { in clkOpd1 result1 } or( 2 ) { out temp };
  { in temp } not( 1 ) { out result2 };
  ```

Component declarations and include statements can only occur at the global level. All other kinds of declarations and statements can also occur within the body of a component declaration or control statement.

Include statements

```
     include "LIBRARY/shift.lib";
```

An include statement has the syntax

```
"include" STRINGLITERAL ";"
```

The STRINGLITERAL represents a file name. The path is relative to the main directory for the assignment.

The file "LIBRARY/globalLibrary.in" is in fact automatically included, and contains declarations of the built-in library components, and also the paths "set" and "clear".

Value Declarations

```
     define base = 16, n = 8, m = 2;
```

A value declaration has the syntax

```
"define" InitValueDefnList ";"
```

where InitValueDefnList is a "," separated list of one or more InitValueDefns. An InitValueDefn has the syntax

```
IDENT "=" Expr
```

At run time, the IDENT is declared in the current run time environment, with value the evaluated expression.

Path Declarations

```
     path opd2, clkOpd1, clkOpd2, result2;
     path result[ n ];
```

A path declaration has the syntax

```
"path" PathDefnList ";"
```

where PathDefnList is a "," separated list of one or more PathDefns.

A PathDefn has the syntax

```
IDENT
```

or

```
IDENT "[" Expr "]"
```

At run time, the IDENT is declared in the current run time environment, with value a path array of the size of the evaluated expression (or 1, if omitted).

If Statements

An if statement is of the form

```
"if" Expr
"then"
    LocalDeclStmtList
"elif" Expr
"then"
    LocalDeclStmtList
...
"else"
    LocalDeclStmtList
"end"
```

There can be zero or more "elif" portions, and the "else" part can be omitted.

At run time, appropriate expressions are evaluated to determine the LocalDeclStmtList to be executed. A local runtime environment is created to execute the LocalDeclStmtList, and declarations within the local block are local to this runtime environment.

For Statements

A for statement is of the form

```
for IDENT from Expr upto Expr
do
    LocalDeclStmtList
end
```

At run time, the expressions are evaluated, and the loop executed with the IDENT having integer values from that of the first expression up to, but not including the second expression.

A local runtime environment is created each time through the loop, and the IDENT is declared in this environment, with an appropriate value. Any declarations in the LocalDeclStmtList are local to this runtime environment.

Component Declarations

```
    component { in opd0[ n ], opd1[ n ] } orBits( n ) { out result[ n ] }
        begin
            for i from 0 upto n do
                { in opd1[ i ] opd0[ i ]  } or( 2 ) { out result[ i ] };
            end
        end
```

A component declaration has the syntax

```
"component" "{" "in" PathDefnList "}"
    IDENT "(" ValueParamDefnList ")"
    "{" "out" PathDefnList "}"
    Body
```

The input path parameter declaration "{ in ... }", value parameter declaration "(...)", and output path parameter declaration "{ out ... }" may each be omitted. A ValueParamDefnList is just a "," separated list of IDENTs.

The body of the component declaration is of the form

```
"begin"
    LocalDeclStmtList
"end"
```

for non-library component declarations, or just

```
";"
```

for library component declarations (which only occur in "LIBRARY/globalLibrary.in").

At run time, the IDENT is declared in the current run time environment, with value a reference to the tree for the declaration, together with the reference to the current environment.

Component Invocation Statements

```
    { in clkOpd1 result1 } or( 2 ).not( 1 ) { out result2 };
```

A component invocation has the syntax

```
"{" "in" PathArrayList "}"
    IDENT "(" ExprList ")" "." ... IDENT "(" ExprList ")"
    "{" "out" PathArrayList "}" ";"
```

The input path parameters "{ in ... }", value parameters "(...)", and output path parameters "{ out ... }" may each be omitted.

The "." separated list of

```
IDENT "(" ExprList ")"
```

represents a "pipelined" list of invocations, where the output paths from a preceding invocation become the input paths of the following invocation. Normally there is only one invocation.

At run time, the input and output parameters at each end of the pipeline are evaluated in the current environment.

Then for each invocation

- The identifier is searched for in the current environment to obtain the declaration of the component, and the environment it was declared in (in fact always the global environment).
- The value parameters are evaluated in the current environment.
- For the first invocation in the pipeline, the input parameters are the input parameters at the start of the pipeline. For other invocations, the input parameters are the output parameters from the preceding invocation in the pipeline.
- For the last invocation in the pipeline, the output parameters are the output parameters at the end of the pipeline. For other invocations, the output path parameters will be created as they are declared, with path array sizes compatible with the invocation.
- A new local environment is created, with enclosing environment the environment in which the component is declared (in fact always the global environment).
- The value parameters are declared in the local environment, with appropriate values.
- The input and output path parameters are declared in the local environment, with appropriate values (the values already exist, except for the output path parameters for a pipelined invocation, for which they are created). The size of the actual path parameter is checked, to ensure it is compatible with the declaration.
- The body of the component declaration is evaluated, using the local environment. For library components, the body is built-in Java code.

Expressions

An expression can involve

```
..."?"...":"... (conditional expression)
"||"
"&&"
"<", "<=", "==", ">", ">="
"<<", ">>"
"+", "-", "&" (string concatenation)
"*", "/", "%"
(unary) "-", "!"
"("...")", IDENT, BINLITERAL, DECLITERAL, HEXLITERAL, STRINGLITERAL, "true",
"false".
```

Expressions only involve integer, Boolean and string variables. You cannot evaluate path variables in an expression. Path values only exist when the logic circuit is simulated.

For string concatenation, the operands are converted to strings before performing concatenation.

Path Expressions

A path array can be declared by a path declaration, such as

```
path carry[ n + 1 ];
```

We may refer to a path or path array, by just writing the name of the path variable, as in

```
{ in opd1 } not( 1 ) { out notOpd1 };
```

We may refer to an element of a path array, by writing "IDENT[index]", as in

```
{ in opd1[ i ], opd2[ i ], carry[ i ] } fullAdder
     { out sum[ i ], carry[ i + 1 ] };
```

We may refer to a subarray, by writing "IDENT[size @ index]", as in

```
{ in opd1[ n - n / 2 @ n / 2 ], opd2[ n - n / 2 @ n / 2 ] }
     compare( n - n / 2 )
     { out lessHigh, equalHigh, greaterHigh };
```

The path name "IDENT[size @ index]" refers to the elements "IDENT[index]", "IDENT[index + 1]", ... "IDENT[index + size - 1]".

Paths can be combined to form a path array, by juxtaposing them, as in

```
{ in cond, opd[ n - p @ 0 ] zero opd } select( 1, n ) { out result };
```

The elements on the left are interpreted as having a high index, while those on the right have the low index. I did it this way because a path array often represents the bits of an integer value, and it is normal to write integers in big-endian format.

The Runtime Environment and Declarations

A runtime environment is a data structure used to map identifiers to values.

Each runtime environment has a name (the name of the component, for the runtime environment for a component), and contains a list of declarations, and a reference to the enclosing runtime environment.

Each declaration has a kind (one of VALUE, PATH, COMPONENT, VALUEPARAM, INPUTPARAM, OUTPUTPARAM), identifier, and value.

Runtime values can be: integer, Boolean and string values; path array values; or component values. A path array value is an array of simple paths. A simple path is essentially just a Boolean variable. A component value is a (component declaration tree, enclosing environment) pair.

The logic circuit simulator

When a path variable declaration is evaluated, it creates paths. When a built-in library component is invoked, it creates a thread that extends the Component class.

A Component thread loops, performs its action (e.g., recomputing the value of its output paths), then pauses.

```
package builtin;

import runEnv.*;

public abstract class Component extends Thread {

     private boolean changed = false;
     private RunEnv runEnv;
     public RunEnv runEnv() { return runEnv; }
     public Component( RunEnv runEnv ) {
          this.runEnv = runEnv;
          runEnv.addListener( this );
          }

     public void run() {
          while ( true ) {
               compute();
               pause();
               }
          }

     public abstract void compute();

     public synchronized void pause() {
          if ( ! changed )
               try {
                    wait();
                    }
               catch ( InterruptedException exception ) {
                    }
          changed = false;
          }

     public synchronized void schedule() {
          changed = true;
          notify();
          }
     }
```

As well as having a value, a simple path has a list of library components that have that path as an input parameter. If the value of a path is altered, all library components that have that path as an input parameter are scheduled to resume. Components with a GUI interface might also be scheduled to resume if a text field is edited, or button pressed.

Each library component overrides the code to perform its computation.

```
package builtin;

import runEnv.*;

public class AndComponent extends Component {

     public AndComponent( RunEnv runEnv ) {
          super( runEnv );
          }

     public void compute() {
          int n = runEnv().getValue( "n" ).intValue();
          PathArrayValue opd = runEnv().getValue( "opd" ).pathArrayValue();
          PathArrayValue result = runEnv().getValue(
               "result" ).pathArrayValue();
          if ( opd.size() != n )
               throw new Error( "Size of " + opd + " != " + n );
          if ( result.size() != 1 )
               throw new Error( "Size of " + result + " != " + 1 );
          boolean resultValue = true;
          for ( int i = 0; i < n; i++ )
               resultValue &= opd.elementAt( i ).getValue();
          result.elementAt( 0 ).setValue( resultValue );
          }
     }
```

The top level code

The “interpreter” takes a sequence of arguments

- -dir directoryName

  Specifies the name of a directory containing a file “program.in” to execute. A file “program.err” is generated containing any error messages; “program.print” is generated containing a reprinted program, that should be essentially the same as “program.in”, but missing comments, and with different formattting; “program.out” is generated containing information on the components and paths created, and how they are linked together.

- -debug

  Turns on debugging. Use Print.error().debugln() and Print.error().debugStackTrace() to print debugging messages.

- -expand

  Causes the code in “LIBRARY/globalLibrary.in” and include files to be reprinted in “program.print”.

- -print

  Causes the program to be reprinted, but not evaluated.

The “interpreter”

- Lexically analyses and parses the file “LIBRARY/globalLibrary.in” and user program “program.in” and builds a tree, representing the program.
- Reprints the program, using the toString() method that every node of the tree has.
- Invokes the eval() method on the root node of the tree, to interpret the program.

```
package circuit;

import java.io.*;
import java_cup.runtime.*;
import node.*;
import node.stmtNode.*;
import text.*;
import grammar.*;

import runEnv.*;

public class Main {

     public static boolean debug = false;
     public static boolean expand = false;
     public static boolean print = false;

     public static void main( String[] argv ) {
          String dirName = null;

          try {
               for ( int i = 0; i < argv.length; i++ ) {
                    if ( argv[ i ].equals( "-debug" ) ) {
                         debug = true;
                         }
                    else if ( argv[ i ].equals( "-expand" ) ) {
                         expand = true;
                         }
                    else if ( argv[ i ].equals( "-print" ) ) {
                         print = true;
                    }
                else if ( argv[ i ].equals( "-dir" ) ) {
                    i++;
                    if ( i >= argv.length )
                        throw new Error( "Missing directory name" );
                    dirName = argv[ i ];
                    }
                else {
                    throw new Error(
                        "Invalid argument: \"" + argv[ i ]
                        + "\"\nUsage: java Main [-debug] [-expand] [-print]
-dir directory" );
                    }
                }

            if ( dirName == null )
                throw new Error( "Directory not specified" );

            Print.setError( new File( dirName, "program.err" ) );
            Print.setReprint( new File( dirName, "program.print" ) );
            Print.setInterp( new File( dirName, "program.out" ) );

            parser libraryParser = new parser(
                new File( "LIBRARY", "globalLibrary.in" ) );
            DeclStmtListNode globalLibrary =
                ( DeclStmtListNode ) libraryParser.parse().value;

            parser programParser = new parser(
                new File( dirName, "program.in" ) );
            DeclStmtListNode declStmtList =
                ( DeclStmtListNode ) programParser.parse().value;

            ProgramNode program = new ProgramNode(
                globalLibrary, declStmtList );

            Print.error().println( "Reprinting ... " );
            Print.reprint().print( program );
            if ( print )
                System.exit( 0 );
            Print.error().println( "Evaluate ... " );
            try {
                program.eval( new RunEnv( "<main>", null ) );
                }
            catch ( Error exception ) {
                Print.error().println(
                    "User Error " + exception.getMessage() );
                Print.error().printStackTrace( exception );
                }
            }
        catch ( Throwable exception ) {
            exception.printStackTrace();
            Print.error().println( "Exception in Main " + exception );
            Print.error().printStackTrace( exception );
            System.exit( -1 );
            }
        }
    }
```

What you have to do

- Implement binary and hexadecimal integer literals in the lexical analyser, parser and Java code. Allow integer literals to be binary, decimal, or hexadecimal. Binary literals start with “0b” or “0B”, followed by one or more binary digits. Decimal literals are composed of one of more decimal digits. Hexadecimal literals start with “0x” or “0X”, followed by one or more hexadecimal digits. When reprinting the program, integer literals must be displayed in the same base as the original.

(10 marks)

Write the missing portions of the grammar. When not provided, ensure that the abstract syntax tree can be built, and that nodes have a suitable toString() method to regenerate the program correctly.

- Value declarations and value parameter declarations.
- Path declarations and input and output path parameter declarations.
- Component declarations.
- Invocations.
- Actual path parameters.
- Indexing, subarrays, and grouping of paths by juxtaposition.
- If and for statements.

(70 marks)

Implement the eval() method for the missing nodes.

- Value declarations and value parameter declarations.
- Indexing, subarrays, grouping of paths by juxtaposition.
- Pipelining of invocations.
- If and for statements.

(40 marks)

Write programs in the language to implement at least two kinds of sensible logic circuit, preferably ones that you can execute successfully. You might want to add library files into the “LIBRARY” directory. Put the main program in the “StudentPrograms” directory. Document what you have done in the “Documentation/ReadMe.doc” file.

For example:

- Perform subtraction.
- Perform multiplication.
- Perform a signed right shift.
- Perform a left or right rotate.
- Determine the position of the most significant bit in a number.

(10 marks)

If you achieve most of the assignment, and do something extra, beyond the assignment requirements, you can be given up to 10 bonus marks. Before you do this, tell me what you intend to do, and make sure you provide documentation to the marker to tell them what you have done.

(10 marks)

Total 130 Marks

Hand In

- Change into your “ASSIGN2STUD” directory.
- Run the Bash shell script “createtar.bash”, to tar and gzip the relevant files into a gzipped tar file, “ASSIGN2.tar.gz”, representing your complete program.
- Only the “Documentation”, “LIBRARY”, “StudentPrograms” and “Source” subdirectories within your “ASSIGN2STUD” directory, will be tarred and gzipped.
- Check that the file really is a gzipped tar file. For example, the UNIX “file” command tells you the file type. Some versions of the “tar” command do not support the -z option, and you might have to first tar the directories, then gzip them separately.
- Submit the gzipped tar file “ASSIGN2.tar.gz”, using the assignment drop box.

Due Date 12 Noon, Wednesday (week 10) 16th May 2007, subject to normal Bonus/Penalty rules, as described in the introductory handout.

Bruce Hutton